30 | 虚拟DOM（上）：如何通过虚拟DOM更新页面？

2021-12-29 大圣

《玩转Vue 3全家桶》 课程介绍 >

讲述：大圣

时长 08:47 大小 8.06M

你好，我是大圣。

上一讲我们主要介绍了 Vue 项目的首次渲染流程，在 mountComponent 中注册了 effect 函数，这样，在组件数据有更新的时候，就会通知到组件的 update 方法进行更新。

Vue 中组件更新的方式也是使用了响应式 + 虚拟 DOM 的方式，这个我们在第一讲中有介绍过 Vue 1、Vue 2 和 Vue 3 中更新方式的变化，今天我们就来详细剖析一下 Vue 组件内部如何通过虚拟 DOM 更新页面的代码细节。

Vue 虚拟 DOM 执行流程

我们从虚拟 DOM 在 Vue 的执行流程开始讲起。在 Vue 中，我们使用虚拟 DOM 来描述页面的组件，比如下面的 template 虽然格式和 HTML 很像，但是在 Vue 的内部会解析成 JavaScript 函数，这个函数就是用来返回虚拟 DOM：

复制代码

```
<div id="app">
  <p>hello world</p>
  <Rate :value="4"></Rate>
</div>
```

上面的 template 会解析成下面的函数，最终返回一个 JavaScript 的对象能够描述这段 HTML：

复制代码

```
function render(){
  return h('div',{id:"app"},children:[
    h('p',{},'hello world'),
    h(Rate,{value:4}),
  ])
}
```

知道虚拟 DOM 是什么之后，那么它是怎么创建的呢？

DOM 的创建

我们简单回忆上一讲介绍的 mount 函数，在代码中，我们使用 createVNode 函数创建项目的虚拟 DOM，可以看到 **Vue 内部的虚拟 DOM，也就是 vnode，就是一个对象，通过 type、props、children 等属性描述整个节点**：

复制代码

```
const vnode = createVNode(    (
  rootComponent as ConcreteComponent,
  rootProps
)
function _createVNode() {

  // 处理属性和□class
  if (props) {
    ...
  }
```

```
11
12    // 标记vnode信息
13    const shapeFlag = isString(type)
14      ? ShapeFlags.ELEMENT
15      : __FEATURE_SUSPENSE__ && isSuspense(type)
16      ? ShapeFlags.SUSPENSE
17      : isTeleport(type)
18      ? ShapeFlags.TELEPORT
19      : isObject(type)
20      ? ShapeFlags.STATEFUL_COMPONENT
21      : isFunction(type)
22      ? ShapeFlags.FUNCTIONAL_COMPONENT
23      : 0
24
25    return createBaseVNode(
26      type,
27      props,
28      children,
29      patchFlag,
30      dynamicProps,
31      shapeFlag,
32      isBlockNode,
33      true
34    )
35  }
36
37  function createBaseVNode(type,props,children,...){
38      const vnode = {
39      type,
40      props,
41      key: props && normalizeKey(props),
42      ref: props && normalizeRef(props),
43      children,
44      shapeFlag,
45      patchFlag,
46      dynamicProps,
47        ...
48    } as VNode
49    // 标准化子节点
50    if (needFullChildrenNormalization) {
51      normalizeChildren(vnode, children)
52    } else if (children) {
53      vnode.shapeFlag |= isString(children)
54        ? ShapeFlags.TEXT_CHILDREN
55        : ShapeFlags.ARRAY_CHILDREN
56    }
57    return vnode
58  }componentUpdateFn
```

createVNode 负责创建 Vue 中的虚拟 DOM，而上一讲中我们讲过 mount 函数的核心逻辑就是使用 setupComponent 执行我们写的 <script setup>，使用 setupRenderEffect 监听组件的数据变化。所以我们来到 setupRenderEffect□函数中，去完整地剖析 Vue 中虚拟 DOM 的更新逻辑。

我们给组件注册了 update 方法，这个方法使用 effect 包裹后，当组件内的 ref、reactive 包裹的响应式数据变化的时候就会执行 update 方法，触发组件内部的更新机制。

看下面的代码，在 setupRenderEffect 内部的 componentUpdateFn 中，updateComponentPreRenderer 更新了属性和 slots，并且调用 renderComponentRoot 函数创建新的子树对象 nextTree，然后内部依然是调用 patch 函数。

可以看到，**Vue 源码中的实现首次渲染和更新的逻辑都写在一起，我们在递归的时候如果对一个标签实现更新和渲染，就可以用一个函数实现。**

复制代码

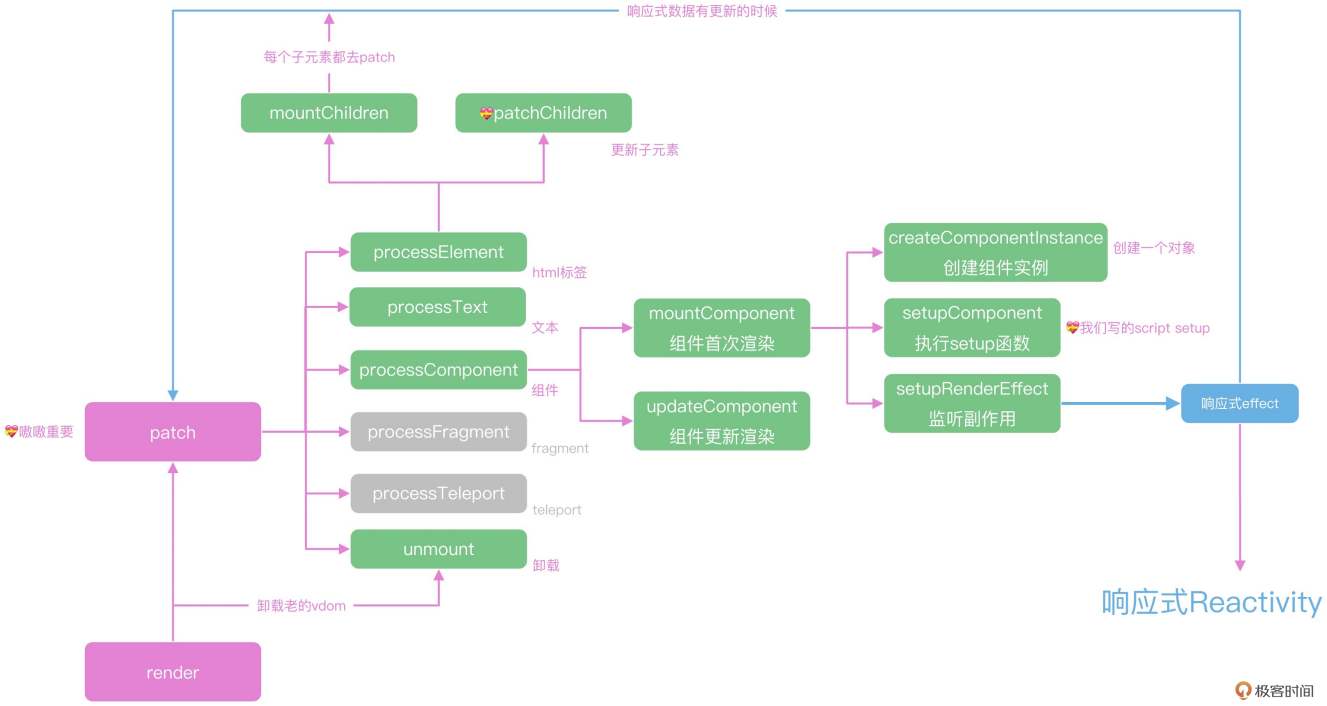
```
const componentUpdateFn = ()=>{
  if (!instance.isMounted) {
      //首次渲染
      instance,
        parentSuspense,
        isSVG
      )
      。。。
  }else{
    let { next, bu, u, parent, vnode } = instance
    if (next) {
      next.el = vnode.el
      updateComponentPreRender(instance, next, optimized)
    } else {
      next = vnode
    }
    const nextTree = renderComponentRoot(instance)
      patch(
        prevTree,
        nextTree,
        // parent may have changed if it's in a teleport
        hostParentNode(prevTree.el!)!,
        // anchor may have changed if it's in a fragment
        getNextHostNode(prevTree),
        instance,
        parentSuspense,
```

```
        isSVG
      )
    }
}

// 注册effect函数
const effect = new ReactiveEffect(
  componentUpdateFn,
  () => queueJob(instance.update),
  instance.scope // track it in component's effect scope
)
const update = (instance.update = effect.run.bind(effect) as S      chedulerJo
update()

  const updateComponentPreRender = (
    instance: ComponentInternalInstance,
    nextVNode: VNode,
    optimized: boolean
  ) => {
    nextVNode.component = instance
    const prevProps = instance.vnode.props
    instance.vnode = nextVNode
    instance.next = null
    updateProps(instance, nextVNode.props, prevProps, optimized)
    updateSlots(instance, nextVNode.children, optimized)

    pauseTracking()
    // props update may have triggered pre-flush watchers.
    // flush them before the render update.
    flushPreFlushCbs(undefined, instance.update)
    resetTracking()
  }
```

比较关键的就是上面代码中 32-39 行的 **effect 函数，负责注册组件，这个函数也是 Vue 组件更新的入口函数。**

patch 函数

数据更新之后就会执行 patch 函数，下图就是 patch 函数执行的逻辑图：



在 patch 函数中，会针对不同的组件类型执行不同的函数，组件我们会执行 processComponent，HTML 标签我们会执行 processElement：

复制代码

```
function path(n1, n2, container){
  const { type, shapeFlag } = n2
  switch (type) {
    case Text:
      processText(n1, n2, container)
      break
    // 还有注释，fragment之类的可以处理，这里忽略
    default:
      // 通过shapeFlag判断类型
      if (shapeFlag & ShapeFlags.ELEMENT) {
        processElement(n1, n2, container, anchor)
      } else if (shapeFlag & ShapeFlags.STATEFUL_COMPONENT) {
        processComponent(n1, n2, container)
      }
  }

}

function processComponent(n1, n2, container) {
  // 老规矩，么有n1就是mount
  if (!n1) {
    // 初始化 component
    mountComponent(n2, container)
  } else {
    updateComponent(n1, n2, container)
  }
```

```
27   }
```

由于更新之后不是首次渲染了，patch 函数内部会执行 updateComponent，看下面的 updateComponent 函数内部，shouldUpdateComponent 会判断组件是否需要更新，实际执行的是 instance.update：

复制代码

```
const instance = (n2.component = n1.component)!
if (shouldUpdateComponent(n1, n2, optimized)) {

  // normal update
  instance.next = n2
  // in case the child component is also queued, remove it to avoid
  // double updating the same child component in the same flush.
  invalidateJob(instance.update)
  // instance.update is the reactive effect.
  instance.update()

} else {
  // no update needed. just copy over properties
  n2.component = n1.component
  n2.el = n1.el
  instance.vnode = n2
}
```

组件的子元素是由 HTML 标签和组件构成，组件内部的递归处理最终也是对 HTML 标签的处理，所以，最后组件的更新都会进入到 processElement 内部的 patchElement 函数中。

patchElement 函数

在函数 patchElement 中我们主要就做两件事，更新节点自己的属性和更新子元素。

节点自身属性的更新

先看自身属性的更新，这里就能体现出 **Vue 3 中性能优化的思想，通过 patchFlag 可以做到按需更新**：

如果标记了 FULL_PROPS，就直接调用 patchProps。

如果标记了 CLASS，说明节点只有 class 属性是动态的，其他的 style 等属性都不需要进行判断和 DOM 操作。

这样就极大的优化了属性操作的性能。

内部执行 hostPatchProp 进行实际的 DOM 操作，你还记得上一讲中 hostPatchProp 是从 nodeOps 中定义的吗，其他动态属性 STYLE、TEXT 等等也都是一样的逻辑。Vue 3 的虚拟 DOM 真正做到了按需更新，这也是相比于 React 的一个优势。

复制代码

```
const patchElement = (
  n1: VNode,
  n2: VNode,
  parentComponent: ComponentInternalInstance | null,
  parentSuspense: SuspenseBoundary | null,
  isSVG: boolean,
  slotScopeIds: string[] | null,
  optimized: boolean
) => {
  const el = (n2.el = n1.el!)
  let { patchFlag, dynamicChildren, dirs } = n2
  patchFlag |= n1.patchFlag & PatchFlags.FULL_PROPS

  const oldProps = n1.props || EMPTY_OBJ
  const newProps = n2.props || EMPTY_OBJ

  // full diff
  patchChildren(
    n1,
    n2,
    el,
    null,
    parentComponent,
    parentSuspense,
    areChildrenSVG,
    slotScopeIds,
    false
  )

  if (patchFlag > 0) {

    if (patchFlag & PatchFlags.FULL_PROPS) {
      patchProps(
        el,
        n2,
        oldProps,
        newProps,
```

```
          parentComponent,
          parentSuspense,
          isSVG
        )
      } else {
        // class是动态的
        if (patchFlag & PatchFlags.CLASS) {
          if (oldProps.class !== newProps.class) {
            hostPatchProp(el, 'class', null, newProps.class, isSVG)
          }
        }

        // style样式是动态的
        if (patchFlag & PatchFlags.STYLE) {
          hostPatchProp(el, 'style', oldProps.style, newProps.style, isSVG)
        }

        // 属性需要diff
        if (patchFlag & PatchFlags.PROPS) {
          //
          const propsToUpdate = n2.dynamicProps!
          for (let i = 0; i < propsToUpdate.length; i++) {
            const key = propsToUpdate[i]
            const prev = oldProps[key]
            const next = newProps[key]
            // #1471 force patch value
            if (next !== prev || key === 'value') {
              hostPatchProp(
                el,
                key,
                prev,
                next,
                isSVG,
                n1.children as VNode[],
                parentComponent,
                parentSuspense,
                unmountChildren
              )
            }
          }
        }
        //文本是动态的
        if (patchFlag & PatchFlags.TEXT) {
          if (n1.children !== n2.children) {
            hostSetElementText(el, n2.children as string)
          }
        }
      }
    }
```

子元素的更新

而子元素的更新是 patchChildren 函数负责的，这个函数也是虚拟 DOM 中难度最高的一个函数，搞懂它还需要我们下一讲中介绍的算法知识，今天我们就先理解它主要的实现思路。

首先我们把子元素分成了文本、数组和空三个状态，新老子元素分别是这三种状态的一个，构成了不同的执行逻辑。这样 patchChildren 内部大致有五种情况需要处理：

如果新的子元素是空， 老的子元素不为空，直接卸载 unmount 即可。

如果新的子元素不为空，老的子元素是空，直接创建加载即可。

如果新的子元素是文本，老的子元素如果是数组就需要全部 unmount，是文本的话就需要执行 hostSetElementText。

如果新的子元素是数组，比如是使用 v-for 渲染出来的列表，老的子元素如果是空或者文本，直接 unmout 后，渲染新的数组即可。

最复杂的情况就是新的子元素和老的子元素都是数组。

最朴实无华的思路就是把老的子元素全部 unmount，新的子元素全部 mount，这样虽然可以实现功能，但是没法复用已经存在的 DOM 元素，比如我们只是在数组中间新增了一个数据，全部 DOM 都销毁就有点太可惜了。

所以，我们需要判断出可以复用的 DOM 元素，如果一个虚拟 DOM 没有改动或者属性变了，不需要完全销毁重建，而是更新一下属性，最大化减少 DOM 的操作，这个任务就会交给 patchKeyedChildren 函数去完成。

patchKeyedChildren 函数，做的事情就是尽可能高效地把老的子元素更新成新的子元素，如何高效复用老的子元素中的 DOM 元素是 patchKeyedChildren 函数的难点：

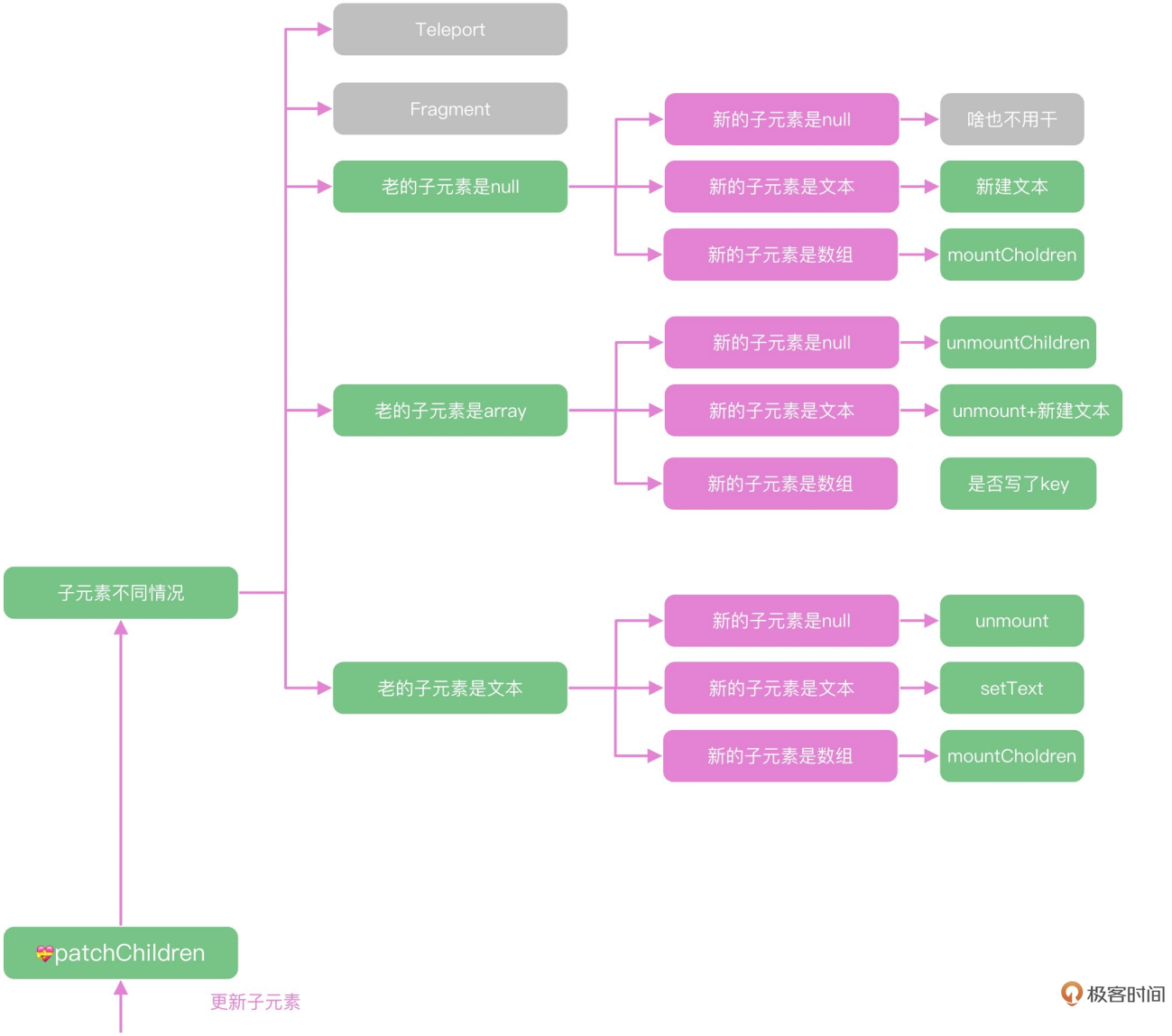
复制代码

```
const patchChildren: PatchChildrenFn = (
  n1,
  n2,
  container,
  anchor,
  parentComponent,
```

```
    parentSuspense,
    isSVG,
    slotScopeIds,
    optimized = false
  ) => {
    const c1 = n1 && n1.children
    const prevShapeFlag = n1 ? n1.shapeFlag : 0
    const c2 = n2.children

    const { patchFlag, shapeFlag } = n2
    // fast path
    if (patchFlag > 0) {
      if (patchFlag & PatchFlags.KEYED_FRAGMENT) {
        // this could be either fully-keyed or mixed (some keyed some not)
        // presence of patchFlag means children are guaranteed to be arrays
        patchKeyedChildren(
          c1 as VNode[],
          c2 as VNodeArrayChildren,
          container,
          anchor,
          parentComponent,
          parentSuspense,
          isSVG,
          slotScopeIds,
          optimized
        )
        return
      } else if (patchFlag & PatchFlags.UNKEYED_FRAGMENT) {
        // unkeyed
        patchUnkeyedChildren(
          c1 as VNode[],
          c2 as VNodeArrayChildren,
          container,
          anchor,
          parentComponent,
          parentSuspense,
          isSVG,
          slotScopeIds,
          optimized
        )
        return
      }
    }

    // children has 3 possibilities: text, array or no children.
    if (shapeFlag & ShapeFlags.TEXT_CHILDREN) {
      // text children fast path
      if (prevShapeFlag & ShapeFlags.ARRAY_CHILDREN) {
        unmountChildren(c1 as VNode[], parentComponent, parentSuspense)
      }
      if (c2 !== c1) {
        hostSetElementText(container, c2 as string)
```

```
      }
    } else {
      if (prevShapeFlag & ShapeFlags.ARRAY_CHILDREN) {
        // prev children was array
        if (shapeFlag & ShapeFlags.ARRAY_CHILDREN) {
          // two arrays, cannot assume anything, do full diff
          patchKeyedChildren(
            c1 as VNode[],
            c2 as VNodeArrayChildren,
            container,
            anchor,
            parentComponent,
            parentSuspense,
            isSVG,
            slotScopeIds,
            optimized
          )
        } else {
          // no new children, just unmount old
          unmountChildren(c1 as VNode[], parentComponent, parentSuspense, true
        }
      } else {
        // prev children was text OR null
        // new children is array OR null
        if (prevShapeFlag & ShapeFlags.TEXT_CHILDREN) {
          hostSetElementText(container, '')
        }
        // mount new if array
        if (shapeFlag & ShapeFlags.ARRAY_CHILDREN) {
          mountChildren(
            c2 as VNodeArrayChildren,
            container,
            anchor,
            parentComponent,
            parentSuspense,
            isSVG,
            slotScopeIds,
            optimized
          )
        }
      }
    }
  }
```

上面的代码执行逻辑如下图所示，根据 flags 判断子元素的类型后，执行不同的操作函数：



patchChildren

最后就剩下 patchChildren 的实现了，这也是各类虚拟 DOM 框架中最难实现的函数，我们需要实现**一个高效的更新算法，能够使用尽可能少的更新次数，来实现从老的子元素到新的子元素的更新。**

举个例子，类似体育课站队的时候，大家一开始站一排，但是顺序是乱的，我们需要尽快把队伍按照个头左低右高排列。

在 React 中，这种场景的处理逻辑是先进行循环，使用的是单侧插入的算法，我们在排队的时候挨个对比，如果你站我右边，并且个头比我高一点，说明咱俩的相对位置和最终队伍的位置是一致的，暂时不需要变化，如果你比我个头矮，就需要去我左边找到一个正确的位置插队进去。

由于都只向单侧插入，最后我们就会把所有的节点移动到正确的位置之上，这就是 React15 框架内虚拟节点 diff 的逻辑，初步实现了 DOM 的复用；而 Vue 2 借鉴了 snabbdom 的算法，在此基础上做了第一层双端对比的优化。

首先 Web 场景之下对一个数组元素的操作，很少有直接全部替换的，**比如我们操作一个表格，大概率是更关心表格某一行的一个字段、新增一行、删除一行，或者是对表格某个字段进行排序，所以我们可以从纯算法的场景之中加入实际应用的场景。**

如果我们只是在表格里新增一行，那么可以不要一开始就开始循环，而是可以先进行节点的预判。

比如，在下面的例子中，新的节点就是在老的节点中新增和删除了几个元素，我们在循环之前，先进行头部元素的判断。在这个例子里，可以预判出头部元素的 a、b、c、d 是一样的节点，说明节点不需要重新创建，我们只需要进行属性的更新，然后进行队尾元素的预判，可以判断出 g 和元素也是一样的：

复制代码

```
1 a b c d e f g h
2 a b c d i f j g h
```

这样我们虚拟 DOM diff 的逻辑就变成了下面的结构, 现在只需要比较 ef 和 ifg 的区别：

复制代码

```
1 (a b c d) e f (g h)
2 (a b c) d) i f j (g h)
```

相比于之前的对比场景，我们需要遍历的运算量就大大减小了。

而且，有很多场景比如新增一行或者删除一行的简单场景，预判完毕之后，新老元素有一个处于没有元素的状态，我们就可以直接执行 mount 或者 unmout 完成对比的全过程，不需要再进行复杂的遍历：

复制代码

```
1 (a b c d)
2 (a b c d) e
```

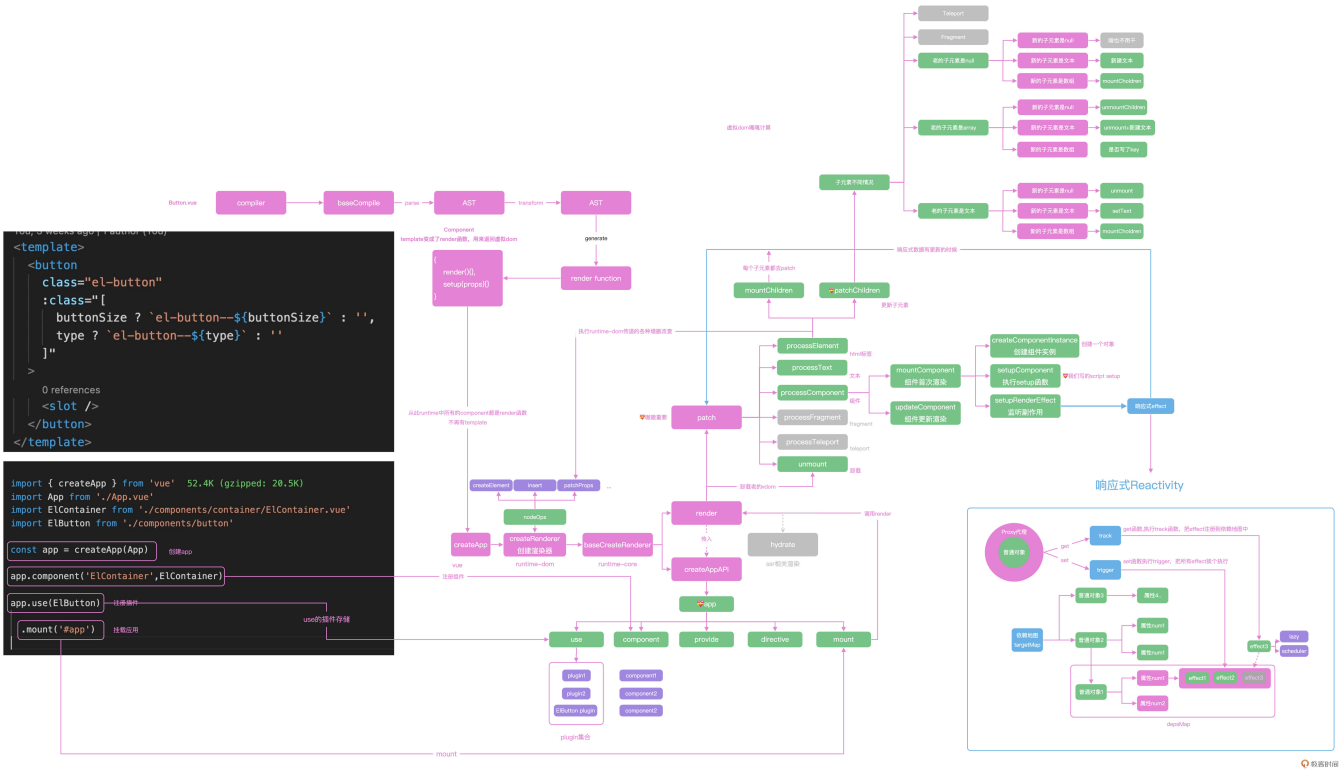
```
3
4 （a b c）d
5 （a b c
```

双端对比的原理大致就是这样。最后双端对比之后的执行逻辑这一部分需要一些算法知识，我们下一讲会详细介绍，这里你只需要掌握大概的思路。

想让一个队伍尽快按照个头排好序，如果能够计算出，在队伍中，个头从低到高依次递增的最多的队列，让这些人站在原地不动，其余人穿插到他们中间，就可以最大化减少人员的移动，这就是一个最长底层子序列的算法问题，我们下一讲详细剖析。

总结

今天的内容就讲完了，来总结一下吧，我们学习了 Vue 中的更新逻辑。现在 Vue 执行逻辑全景图变成了下面的样子，新增了组件更新的逻辑：



Vue 响应式驱动了组件之间的数据通信机制，数据更新之后，组件会执行 intance.update 方法，update 方法内部执行 patch 方法进行新老子树的 diff 计算。

在更新函数中，主要做了两件事，pathProps 更新节点自身的属性，这里面使用了 pathFlags 做到了按需更新；patchChildren 执行子元素的更新。其中 patch 函数内部会

只对节点内部的动态属性做更新，这种按需更新的机制是 Vue 性能优秀的一个原因。

函数内部针对新老子元素不同的状态，执行不同的逻辑。根据子元素是否为空或者数组，以及新元素是否为空或者数组，分别执行对应的删除或者 mount 逻辑，其中最复杂的就是新的子元素和老的子元素都是数组。

为了最大化减少 DOM 操作，patchKeyedChildren 使用了最长递增子序列来实现，并且相比于 React 的虚拟 DOM diff，新增了双端的预先判断 + 最长递增子序列算法来实现，这也是 Vue 性能比较优秀的另外一个原因。

思考题

最后再留一个思考题，从虚拟 DOM 更新的角度，为什么我们在写 v-for 循环的时候，都要建议使用 key 属性呢？欢迎在评论区留下你的答案，也欢迎你把这一讲分享给你的同事和朋友们，我们下一讲再见。

分享给需要的人，Ta订阅后你可得 20 元现金奖励

生成海报并分享

赞 2 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 29 | 运行时：Vue在浏览器里是怎么跑起来的？

下一篇 31 | 虚拟DOM（下）：想看懂虚拟DOM算法，先刷个算法题

更多课程推荐

精选留言 (2)

写留言

Meiki
2021-12-31

vue在进行页面更新时，会通过Diff算法对虚拟DOM进行更新，因为真实的DOM消耗比较大。 假设我们需要对数组进行增删改查，我们需要快速定位到某一项，所以我们需要给每项绑一个具体的唯一的id值

费城的二鹏
2021-12-29

key属性有利于判断新旧列表的复用判断，提升列表乱序后的复用率。